第1章 Hello, World

如果第一个程序员是一个山顶洞人，它在山洞壁 (第一台计算机)上凿出的第一个程序应该是用羚羊图案构成的一个字符串“ Hello, World ”。罗马的编程教科书也应该是以程序“ Salut, Mundi ”开始的。我不知道如果打破这个传统会带来什么后果，至少我还没有勇气去做第一个吃螃蟹的人。

内核模块至少必须有两个函数：init_module和cleanup_module。第一个函数是在把模块插入内核时调用的；第二个函数则在删除该模块时调用。一般来说， init_module可以为内核的某些东西注册一个处理程序，或者也可以用自身的代码来取代某个内核函数 (通常是先干点别的什么事，然后再调用原来的函数)。函数cleanup_module的任务是清除掉init_module所做的一切，这样，这个模块就可以安全地卸载了。

```
ex hello.c

/* hello.c
 * Copyright (C) 1998 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work
*/
#include <linux/module.h>   /* Specifically, a module */
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");

  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}
```

```
/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
```

1.1 内核模块的Makefiles文件

内核模块并不是一个独立的可执行文件，而是一个对象文件，在运行时内核模块被链接到内核中。因此，应该使用 -c命令参数来编译它们。还有一点需要注意，在编译所有内核模块时，都将需要定义好某些特定的符号。

- _ _KERNEL_ _—— 这个符号告诉头文件：这个程序代码将在内核模式下运行，而不要作为用户进程的一部分来执行。
- MODULE —— 这个符号告诉头文件向内核模块提供正确的定义。
- LINUX —— 从技术的角度讲，这个符号不是必需的。然而，如果程序员想要编写一个重要的内核模块，而且这个内核模块需要在多个操作系统上编译，在这种情况下，程序员将会很高兴自己定义了LINUX这个符号。这样一来，在那些依赖于操作系统的部分，这个符号就可以提供条件编译了。

还有其它的一些符号，是否包含它们要取决于在编译内核时使用了哪些命令参数。如果用户不太清楚内核是怎样编译的，可以查看文件 /usr/include/linux/config.h。

- _ _SMP_ _—— 对称多处理。如果编译内核的目的是为了支持对称多处理，在编译时就需要定义这个符号 (即使内核只是在一个 CPU上运行也需要定义它)。当然，如果用户使用对称多处理，那么还需要完成其它一些任务 (参见第12章)。
- CONFIG_MODVERSIONS —— 如果CONFIG_MODVERSIONS可用，那么在编译内核模块时就需要定义它，并且包含头文件 /usr/include/linux/modversions.h。还可以用代码自身来完成这个任务。

```
ex Makefile

# Makefile for a basic kernel module

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: hello.c /usr/include/linux/version.h
        $(CC) $(MODCFLAGS) -c hello.c
        echo insmod hello.o to turn it on
        echo rmmod hello to turn if off
        echo
        echo X and kernel programming do not mix.
        echo Do the insmod and rmmod from outside X.
```

完成了以上这些任务以后，剩下唯一要做的事就是切换到根用户下 (你不是以root身份编译内核模块的吧？别玩什么惊险动作哟！)，然后根据自己的需要插入或删除 hello模块。在执行完insmod命令以后，可以看到新的内核模块在 /proc/modules中。

顺便提一下，Makefile建议用户不要从X执行insmod命令的原因在于，当内核有个消息需要使用printk命令打印出来时，内核会把该消息发送给控制台。当用户没有使用 X时，该消息

将发送到用户正在使用的虚拟终端 (用户可以用Alt-F<n>来选择当前终端)，然后用户就可以看到这个消息了。而另一方面，当用户使用 X时，存在两种可能性。一种情况是用户用命令xterm -C打开了一个控制台，这时输出将被发送到那个控制台；另一种情况是用户没有打开控制台，这时输出将送往虚拟终端7 —— 被X所“覆盖”的一个虚拟终端。

当用户的内核不太稳定时，没有使用 X的用户更有可能取得调试信息。如果没有使用 X，printk将直接从内核把调试消息发送到控制台。而另一方面，在X中printk的消息将被送给一个用户模式的进程(xterm -C)。当那个进程获得CPU时间时，它将把该消息传送给X服务器进程。然后，当X服务器获得CPU时间时，它将显示该消息 —— 但是一个不稳定的内核通常意味着系统将要崩溃或者重新启动，所以用户不希望推迟错误信息显示的时间，因为该信息可能会向用户解释什么地方出了问题，如果显示的时刻晚于系统崩溃或重启的时刻，用户将会错过这个重要的信息。

1.2 多重文件内核模块

有时候在多个源文件间划分内核模块是很有意义的。这时用户需要完成下面三件任务：

1) 除了一个源文件以外，在其它所有源文件中加入一行 #define _ _ NO_VERSION_ _。这点很重要，因为module.h中通常会包含有kernel_version的定义(kernel_version是一个全局变量，它表明该模块是为哪个内核版本所编译的)。如果用户需要version.h文件，那么用户必须自己把它包含在源文件中，因为在定义了 _ _NO_VERSION_ _的情况下， module.h是不会为用户完成这个任务的。

2) 像平常一样编译所有的源文件。

3) 把所有的对象文件组合进一个文件中。在 x86下，可以使用命令：

ld -m elf_i386 -r -o〈模块名称〉.o (第一个源文件).o (第二个源文件).o来完成这个任务。

下面是这种内核模块的一个例子。

ex start.c

```
/* start.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 * This file includes just the start routine
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */


/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");
```

```
  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}
ex stop.c
/* stop.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version. This
 * file includes just the stop routine.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */

#define __NO_VERSION__      /* This isn't "the" file
                             * of the kernel module */
#include <linux/module.h>   /* Specifically, a module */

#include <linux/version.h>   /* Not included by
                              * module.h because
                              * of the __NO_VERSION__ */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
ex Makefile

# Makefile for a multifile kernel module
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o:    start.o stop.o
            ld -m elf_i386 -r -o hello.o start.o stop.o

start.o:    start.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c start.c

stop.o:             stop.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c stop.c
```

第2章　字符设备文件

我们现在就可以吹牛说自己是内核程序员了。虽然我们所写的内核模块还什么也干不了，但我们仍然为自己感到骄傲，简直可以称得上趾高气扬。但是，有时候在某种程度上我们也会感到缺少点什么，简单的模块并不是太有趣。

内核模块主要通过两种方法与进程打交道。一种方法是通过设备文件（例如在目录/dev中的文件），另一种方法是使用proc文件系统。既然编写内核模块的主要原因之一就是支持某些类型的硬件设备，那么就让我们从设备文件开始吧。

设备文件最初的用途是使进程与内核中的设备驱动程序通信，并且通过设备驱动程序再与物理设备(调制解调器、终端等等)通信。下面我们要讲述实现这一任务的方法。

每个设备驱动程序都被赋予一个主编号，主要用于负责某几种类型的硬件。可以在/proc/devices中找到驱动程序以及它们对应的主编号的列表。由设备驱动程序管理的每个物理设备都被赋予一个从编号。这些设备中的每一个，不管是否真正安装在计算机系统上，都将对应一个特殊的文件，该文件称为设备文件，所有的设备文件都包含在目录 /dev中。

例如，如果执行命令 ls -l /dev/hd[ab]*，用户将可以看到与一个计算机相连接的所有的IDE硬件分区。注意，所有的这些硬盘分区都使用同一个主编号： 3，但是从编号却各不相同。需要强调的是，这里假设用户使用的是 PC体系结构。我并不知道在其它体系结构上运行的Linux的设备是怎么样的。

在安装了系统以后，所有的设备文件都由命令 mknod创建出来。从技术的角度上讲，并没有什么特别的原因一定要把这些设备文件放在目录 /dev中，这只不过是一个有用的传统习惯而已。如果读者创建设备文件的目的只不过是为了试试看，就像本章的练习一样，那么把该设备文件放置在编译内核模块的目录中可能会更有意义一些。

设备一般分为两种类型：字符设备和块设备。它们的区别在于块设备具有一个请求缓冲区，所以块设备可以选择按照何种顺序来响应这些请求。这对于存储设备来说是很重要的。在存储设备中，读或写相邻的扇区速度要快一些，而读写相互之间离得较远的扇区则要慢得多。另一个区别在于块设备只能以成块的形式接收输入和返回输出（块的大小根据设备类型的变化而有所不同），而字符设备则可以随心所欲地使用任意数目的字节。当前大多数设备都是字符设备，因为它们既不需要某种形式的缓冲，也不需要按照固定的块大小来进行操作。如果想知道某个设备文件对应的是块设备还是字符设备，用户可以执行命令 ls -l，查看一下该命令的输出中的第一个字符，如果第一个字符是“ b ”，则对应的是块设备；如果是“ c ”，则对应的是字符设备。

模块分为两个独立的部分：模块部分和设备驱动程序部分。前者用于注册设备。函数init_module调用module_register_chrdev，把该设备驱动程序加入到内核的字符设备驱动程序表中，它还会返回供驱动程序所使用的主编号。函数 cleanup_module则取消该设备的注册。

注册某设备和取消它的注册是这两个函数最基本的功能。内核中的东西并不是按照它们自己的意愿主动开始运行的，就像进程一样，而是由进程通过系统调用来调用，或者由硬件

设备通过中断来调用，或者由内核的其它部分调用 (只需调用特定的函数)，它们才会执行。因此，如果用户往内核中加入了代码，就必须把它作为某种特定类型事件的处理程序进行注册；而在删除这些代码时，用户必须取消它的注册。

设备驱动程序一般是由四个 device_<action>函数所组成的，如果用户需要处理具有对应主编号的设备文件，就可以调用这四个函数。通过 file_operations结构Fops内核可以知道调用哪些函数。因为该结构的值是在注册设备时给定的，它包含了指向这四个函数的指针。

在这里我们还需要记住的一点是：无论如何不能乱删内核模块。原因在于如果设备文件是由进程打开的，而我们删去了该内核模块，那么使用该文件就将导致对正确的函数 (读/写) 原来所处的存储位置的调用。如果我们走运，那里没有装入什么其它的代码，那我们至多得到一些难看的错误信息，而如果我们不走运，在原来的同一位置已经装入了另一个内核模块，这就意味着跳转到了内核中另一个函数的中间，这样做的后果是不堪设想的，起码不会是令人愉快的。

一般来说，如果用户不愿意让某件事发生，可以让执行这件事的函数返回一个错误代码 (一个负数)。而对cleanup_module来说这是不可能的，因为它是一个 void函数。一旦调用了 cleanup_module，这个模块就死了。然而，还有一个计数器记录了有多少个其它的内核模块正在使用该内核模块，这个计数器称为引用计数器 (就是位于文件/proc/modules信息行中的最后那个数值)。如果这个数值不为零， rmmod将失败。模块的引用计数值可以从变量 mod_use_count_中得到。因为有些宏是专门为处理这个变量而定义的(如MOD_INC_USE_COUNT和MOD_DEC_USE_COUNT)，我们宁愿使用这些宏，也不愿直接对mod_use_count_进行操作，这样一来，如果将来实现方法有所变化，我们也会很安全。

them, rather than mod_use_count_ directly, so we'll be safe if the implementation c
in the future.

ex chardev.c

```
/* chardev.c
 * Copyright (C) 1998-1999 by Ori Pomerantz
 *
 * Create a character device (read only)
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */
```

```
#include <linux/fs.h>       /* The character device
                             * definitions are here */
#include <linux/wrapper.h>  /* A wrapper which does
                             * next to nothing at
                             * at present, but may
                             * help for compatibility
                             * with future versions
                             * of Linux */


/* In 2.2.3 /usr/include/linux/version.h includes
 * a macro for this, but 2.0.35 doesn't - so I add
 * it here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
/* Conditional compilation. LINUX_VERSION_CODE is
 * the code (as per KERNEL_VERSION) of this version. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for put_user */
#endif



#define SUCCESS 0


/* Device Declarations **************************** */

/* The name for our device, as it will appear
 * in /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message from the device */
#define BUF_LEN 80

/* Is the device open right now? Used to prevent
 * concurent access into the same device */
static int Device_Open = 0;

/* The message the device will give when asked */
static char Message[BUF_LEN];

/* How far did the process reading the message
 * get? Useful if the message is larger than the size
 * of the buffer we get to fill in device_read. */
static char *Message_Ptr;


/* This function is called whenever a process
```

```
 * attempts to open the device file */
static int device_open(struct inode *inode,
                       struct file *file)
{
  static int counter = 0;

#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif

  /* This is how you get the minor device number in
 * case you have more than one physical device using
 * the driver. */
printk("Device: %d.%d\n",
   inode->i_rdev >> 8, inode->i_rdev & 0xFF);

/* We don't want to talk to two processes at the
 * same time */
if (Device_Open)
  return -EBUSY;

/* If this was a process, we would have had to be
 * more careful here.
 *
 * In the case of processes, the danger would be
 * that one process might have check Device_Open
 * and then be replaced by the schedualer by another
 * process which runs this function. Then, when the
 * first process was back on the CPU, it would assume
 * the device is still not open.
 *
 * However, Linux guarantees that a process won't be
 * replaced while it is running in kernel context.
 *
 * In the case of SMP, one CPU might increment
 * Device_Open while another CPU is here, right after
 * the check. However, in version 2.0 of the
 * kernel this is not a problem because there's a lock
 * to guarantee only one CPU will be kernel module at
 * the same time. This is bad in  terms of
 * performance, so version 2.2 changed it.
 * Unfortunately, I don't have access to an SMP box
 * to check how it works with SMP.
 */

Device_Open++;

/* Initialize the message. */
sprintf(Message,
  "If I told you once, I told you %d times - %s",
```

```
    counter++,
    "Hello, world\n");
  /* The only reason we're allowed to do this sprintf
   * is because the maximum length of the message
   * (assuming 32 bit integers - up to 10 digits
   * with the minus sign) is less than BUF_LEN, which
   * is 80. BE CAREFUL NOT TO OVERFLOW BUFFERS,
     * ESPECIALLY IN THE KERNEL!!!
     */

    Message_Ptr = Message;

    /* Make sure that the module isn't removed while
     * the file is open by incrementing the usage count
     * (the number of opened references to the module, if
     * it's not zero rmmod will fail)
     */
    MOD_INC_USE_COUNT;

    return SUCCESS;
  }


  /* This function is called when a process closes the
   * device file. It doesn't have a return value in
   * version 2.0.x because it can't fail (you must ALWAYS
   * be able to close a device). In version 2.2.x it is
   * allowed to fail - but we won't let it.
   */
  #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  static int device_release(struct inode *inode,
                            struct file *file)
  #else
  static void device_release(struct inode *inode,
                             struct file *file)
  #endif
  {
  #ifdef DEBUG
    printk ("device_release(%p,%p)\n", inode, file);
  #endif

    /* We're now ready for our next caller */
    Device_Open --;

    /* Decrement the usage count, otherwise once you
     * opened the file you'll never get rid of the module.
     */
    MOD_DEC_USE_COUNT;

  #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    return 0;
```

```
#endif
}
/* This function is called whenever a process which
 * have already opened the device file attempts to
 * read from it. */


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(struct file *file,
    char *buffer,    /* The buffer to fill with data */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
#else
static int device_read(struct inode *inode,
                       struct file *file,
    char *buffer,   /* The buffer to fill with
                     * the data */
    int length)     /* The length of the buffer
                     * (mustn't write beyond that!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;

  /* If we're at the end of the message, return 0
   * (which signifies end of file) */
  if (*Message_Ptr == 0)
    return 0;

  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {

    /* Because the buffer is in the user data segment,
     * not the kernel data segment, assignment wouldn't
     * work. Instead, we have to use put_user which
     * copies data from the kernel data segment to the
     * user data segment. */
    put_user(*(Message_Ptr++), buffer++);


    length --;
    bytes_read ++;
  }

#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif
 /* Read functions are supposed to return the number
  * of bytes actually inserted into the buffer */
```

```
  return bytes_read;
}



/* This function is called when somebody tries to write
 * into our device file - unsupported in this example. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
    const char *buffer,    /* The buffer */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  return -EINVAL;
}



/* Module Declarations ***************************** */

/* The major device number for the device. This is
 * global (well, static, which in this context is global
 * within this file) because it has to be accessible
 * both for registration and for release. */
static int Major;

/* This structure will hold the functions to be
 * called when a process does something to the device
 * we created. Since a pointer to this structure is
 * kept in the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */


struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  NULL,   /* ioctl */
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
```

```
  NULL,   /* flush */
#endif
  device_release  /* a.k.a. close */
};


/* Initialize the module - Register the character device */
int init_module()
{
  /* Register the character device (atleast try) */
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
                                 &Fops);

  /* Negative values signify an error */
  if (Major < 0) {
    printk ("%s device failed with %d\n",
        "Sorry, registering the character",
        Major);
    return Major;
  }

  printk ("%s The major device number is %d.\n",
          "Registeration is a success.",
          Major);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod <name> c %d <minor>\n", Major);
  printk ("You can try different minor numbers %s",
          "and see what happens.\n");

  return 0;
}


/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;
  /* Unregister the device */
  ret = module_unregister_chrdev(Major, DEVICE_NAME);

  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in unregister_chrdev: %d\n", ret);
}
```

多内核版本源文件

系统调用是内核提供给进程的主要接口，它并不随着内核版本的变化而有所改变。当然

可能会加入新的系统调用，但是老的系统调用是永远不会改变的。这主要是为了提供向后兼容性的需要 —— 新的内核版本不应该使原来工作正常的进程出现问题。在大多数情况下，设备文件也应该保持不变。另一方面，内核里面的内部接口则可以变化，并且也确实随着内核版本的变化而改变了。

Linux内核的版本可以划分为稳定版本(n.<偶数>.m)和开发版本(n.<奇数>.m)两种。开发版本中包括所有最新最酷的思想，当然其中也可能有一些将来会被认为是馊主意的错误，有些将会在下一个版本中重新实现。因此，用户不能认为在这些版本之间接口也会保持一致 (这就是我为什么懒得在本书中介绍它们的原因，这需要做大量的工作，而且很快就会过时被淘汰)。另一方面，在稳定的版本中，我们可以无视错误修正版本 (带数字m的版本)而认为接口是保持不变的。

这个MPG版本包含对Linux内核版本2.0.x和版本2.2.x的支持。因为这两个版本之间存在差异，这就要求用户根据内核版本号来进行条件编译。为了做到这一点，可以使用宏LINUX_VERSION_CODE。在内核版本a.b.c中，该宏的值将会是$2^{16}a+2^{8}b+c$。为了获取特定内核版本的值，我们可以使用宏 KERNEL_VERSION。在2.0.35中该宏没有定义，如果需要的话我们可以自己定义它。

第3章　/proc文件系统

在Linux中，内核和内核模块还可以通过另一种方法把信息发送给进程，这种方法就是/proc 文件系统。最初/proc 文件系统是为了可以轻松访问有关进程的信息而设计的 (这就是它的名称的由来)，现在每一个内核部分只要有些信息需要报告，都可以使用 /proc 文件系统，例如/proc /modules包含一个模块的列表，/proc /meminfo包含有关内存使用的统计信息。

使用/proc 文件系统的方法其实与使用设备驱动程序的方法是非常类似的 —— 用户需要创建一个结构，该结构包含了 /proc文件所需要的所有信息，包括指向任意处理程序函数的指针 (在我们本章的例子中只有一个处理程序函数，当有人试图读 /proc文件时将调用这个函数)。然后，init_module将向内核注册这个结构，而 cleanup_module将取消它的注册。

在程序中之所以需要使用 proc_register_dynamic，是因为我们不想事先判断文件所使用的索引节点编号，而是让内核去决定，这样可以避免编号冲突。一般文件系统都是位于磁盘上的，而不是仅仅存在于内存中 (/proc是存在于内存中的)，在那种情况下，索引节点编号是一个指向某个磁盘位置的指针，在那个位置上存放了该文件的索引节点 (简写为inode)。索引节点包含该文件的有关信息，例如文件的访问权限，以及指向某个或者某些磁盘位置的指针，在这个或者这些磁盘位置中，存放着文件的数据。

因为在文件打开或者关闭时该模块不会被调用，所以我们无需在该模块中使用MOD_INC_USE_COUNT和MOD_DEC_USE_COUNT。如果文件打开以后模块被删除了，没有任何措施可以避免这一后果。在下一章中，我们将学习到一种比较难于实现，但却相对方便的方法，可以用于处理/proc文件，我们也可以使用那个方法来防止这个问题。

```
ex procfs.c

/* procfs.c -  create a "file" in /proc
 * Copyright (C) 1998-1999 by Ori Pomerantz
 */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Necessary because we use the proc fs */
```

```
#include <linux/proc_fs.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Put data into the proc fs file.

   Arguments
   =========
   1. The buffer where the data is to be inserted, if
      you decide to use it.
   2. A pointer to a pointer to characters. This is
      useful if you don't want to use the buffer
      allocated by the kernel.
   3. The current position in the file.
   4. The size of the buffer in the first argument.
   5. Zero (for future use?).


   Usage and Return Value
   ======================
   If you use your own buffer, like I do, put its
   location in the second argument and return the
   number of bytes used in the buffer.

   A return value of zero means you have no further
   information at this time (end of file). A negative
   return value is an error condition.


   For More Information
   ====================
   The way I discovered what to do with this function
   wasn't by reading documentation, but by reading the
   code which used it. I just looked to see what uses
   the get_info field of proc_dir_entry struct (I used a
   combination of find and grep, if you're interested),
   and I saw that  it is used in <kernel source
   directory>/fs/proc/array.c.

   If something is unknown about the kernel, this is
   usually the way to go. In Linux we have the great
   advantage of having the kernel source code for
```

```
     free - use it.
   */
  int procfile_read(char *buffer,
                    char **buffer_location,
                    off_t offset,
                    int buffer_length,
                    int zero)
  {
    int len;  /* The number of bytes actually used */

    /* This is static so it will still be in memory
     * when we leave this function */
    static char my_buffer[80];

    static int count = 1;
  /* We give all of our information in one go, so if the
   * user asks us if we have more information the
   * answer should always be no.
   *
   * This is important because the standard read
   * function from the library would continue to issue
   * the read system call until the kernel replies
   * that it has no more information, or until its
   * buffer is filled.
   */
  if (offset > 0)
    return 0;

  /* Fill the buffer and get its length */
  len = sprintf(my_buffer,
    "For the %d%s time, go away!\n", count,
    (count % 100 > 10 && count % 100 < 14) ? "th" :
      (count % 10 == 1) ? "st" :
        (count % 10 == 2) ? "nd" :
          (count % 10 == 3) ? "rd" : "th" );
  count++;

  /* Tell the function which called us where the
   * buffer is */
  *buffer_location = my_buffer;

  /* Return the length */
  return len;
}


struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
    4, /* Length of the file name */
```

```
"test", /* The file name */
S_IFREG | S_IRUGO, /* File mode - this is a regular
                    * file which can be read by its
                    * owner, its group, and everybody
                    * else */
1,      /* Number of links (directories where the
     * file is referenced) */
0, 0,  /* The uid and gid for the file - we give it
        * to root */
80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the inode
           * (linking, removing, etc.) - we don't
           * support any. */
    procfile_read, /* The read function for this file,
                    * the function called when somebody
                    * tries to read something from it. */
    NULL /* We could have here a function to fill the
          * file's inode, to enable us to play with
          * permissions, ownership, etc. */
  };


/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register[_dynamic] is a success,
   * failure otherwise. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
   * inode number automatically if it is zero in the
   * structure, so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* proc_root is the root directory for the proc
   * fs (/proc). This is where we want our file to be
   * located.
   */
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第4章　把/proc用于输入

到目前为止，可以通过两种方法从内核模块产生输出：我们可以注册一个设备驱动程序，并使用mknod命令创建一个设备文件；我们还可以创建一个 /proc文件。这样内核模块就可以告诉我们各种各样的信息。现在唯一的问题是我们没有办法来回答它。如果要把输入信息发送给内核模块，第一个方法就是把这些信息写回到 /proc文件中。

因为编写proc文件系统的主要目的是为了让内核可以把它的状态报告给进程，对于输入并没有提供相应的特别措施。结构 proc_dir_entry中并不包含指向输入函数的指针，它只包含指向输出函数的指针。如果需要输入，为了把信息写到 /proc文件中，用户需要使用标准的文件系统机制。

Linux为文件系统注册提供了一个标准的机制。因为每个文件系统都必须具有自己的函数专门用于处于索引节点和文件操作，所以 Linux提供了一个特殊的结构inode_operations，该结构存放指向所有这些函数的指针，其中包含一个指向结构 file_operations的指针。在/proc中，无论何时注册一个新文件，用户都可以指定使用哪个 inode_operations结构来访问它。这就是我们所使用的机制。结构 inode_operations包含指向结构 file_operations的指针，而结构file_operations又包含指向module_input和module_output函数的指针。

注意　在内核中读和写的标准角色是互换的，读函数用于输出，而写函数则用于输入，记住这点很重要。之所以会这样，是因为读和写实际上是站在用户的观点来说的—如果一个进程从内核读信息，内核需要做的是输出这些信息，而如果进程向内核写信息，内核当然会把它当作输入来接收。

这里另外一个有趣的地方是module_permission函数。只要进程试图对/proc文件干点什么，这个函数就将被调用，它可以判断是允许对文件进行访问，还是拒绝这次访问。目前这种判断还只是基于操作本身以及当前所使用的 uid来作出(当前所使用的 uid可以从current得到，current是一个指针，指向包含有关当前运行进程的信息结构)，但是函数module_permission还可以基于用户所选择的任意条件来作出允许或是拒绝访问的判断，例如其它还有什么进程正在使用这个文件、日期和时间、或者我们最近接收到的输入。

在程序中我们之所以使用put_user和get_user，主要是因为Linux内存是分段的(在Intel体系结构下；有些其它的处理器可能会有所不同)。这就意味着一个指针并不能指向内存中的某个唯一的位置，而只能指向一个内存段。为了能够使用指针，用户必须知道它指向的是哪个内存段。内核只对应一个内存段，且每个进程都对应一个内存段。

进程所能访问的唯一的内存段就是它自己的内存段，所以在写将要当作进程来运行的常规程序时，程序员不需要考虑有关分段的问题，而当用户编写内核模块时，通常用户需要访问内核的内存段，该内存段是由系统自动处理的。然而，如果内存缓冲区中的内容需要在当前运行的进程和内核之间传送，内核函数将会接收到一个指针，该指针指向进程段中的内存缓冲区。宏put_user和get_user使用户可以访问那块内存。

ex procfs.c

```
/* procfs.c -  create a "file" in /proc, which allows
 * both input and output. */

/* Copyright (C) 1998-1999 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
/* Necessary because we use proc fs */
#include <linux/proc_fs.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif

/* The module's file functions ********************** */


/* Here we keep the last message received, to prove
 * that we can process our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];


/* Since we use the file operations struct, we can't
 * use the special proc output provisions - we have to
 * use a standard read function, which is this function */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,    /* The file read */
```

```
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    size_t len,  /* The length of the buffer */
    loff_t *offset) /* Offset in the file - ignore */
#else
static int module_output(
    struct inode *inode, /* The inode read */
    struct file *file,   /* The file read */
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    int len)  /* The length of the buffer */
#endif
{
 static int finished = 0;
 int i;
 char message[MESSAGE_LENGTH+30];

 /* We return 0 to indicate end of file, that we have
  * no more information. Otherwise, processes will
  * continue to read from us in an endless loop. */
 if (finished) {
   finished = 0;
   return 0;
 }

 /* We use put_user to copy the string from the kernel's
  * memory segment to the memory segment of the process
  * that called us. get_user, btw, is
  * used for the reverse. */
 sprintf(message, "Last input:%s", Message);
 for(i=0; i<len && message[i]; i++)
   put_user(message[i], buf+i);


 /* Notice, we assume here that the size of the message
  * is below len, or it will be received cut. In a real
  * life situation, if the size of the message is less
  * than len, then we'd return len and on the second call
  * start filling the buffer with the len+1'th byte of
  * the message. */
 finished = 1;

 return i;  /* Return the number of bytes "read" */
}


/* This function receives input from the user when the
 * user writes to the /proc file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
```

```
    struct file *file,    /* The file itself */
    const char *buf,      /* The buffer with input */
    size_t length,        /* The buffer's length */
    loff_t *offset)       /* offset to file - ignore */
#else
static int module_input(
    struct inode *inode, /* The file's inode */
    struct file *file,    /* The file itself */
    const char *buf,      /* The buffer with the input */
    int length)           /* The buffer's length */
#endif
{
  int i;

  /* Put the input into Message, where module_output
   * will later be able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buf+i);
  /* In version 2.2 the semantics of get_user changed,
   * it not longer returns a character, but expects a
   * variable to fill up as its first argument and a
   * user segment pointer to fill it from as the its
   * second.
   *
   * The reason for this change is that the version 2.2
   * get_user can also read an short or an int. The way
   * it knows the type of the variable it should read
   * is by using sizeof, and for that it needs the
   * variable itself.
   */
#else
    Message[i] = get_user(buf+i);
#endif
  Message[i] = '\0';  /* we want a standard, zero
                       * terminated string */

  /* We need to return the number of input characters
   * used */
  return i;
}



/* This function decides whether to allow an operation
 * (return zero) or not allow it (return a non-zero
 * which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
```

```
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file
 * permissions. The permissions returned by ls -l are
 * for referece only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but
   * only root (uid 0) may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /* If it's anything else, access is denied */
  return -EACCES;
}



/* The file is opened - we don't really care about
 * that, but it does mean we need to increment the
 * module's reference count. */
int module_open(struct inode *inode, struct file *file)
{
  MOD_INC_USE_COUNT;

  return 0;
}


/* The file is closed - again, interesting only because
 * of the reference count. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* success */
#endif
}


/* Structures to register as the /proc file, with
 * pointers to all the relevant functions. ********** */
/* File operations for our proc file. This is where we
```

```
 * place pointers to all the functions called when
 * somebody tries to do something to our file. NULL
 * means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,    /* Somebody opened the file */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* flush, added here in version 2.2 */
#endif
    module_close,    /* Somebody closed the file */
    /* etc. etc. etc. (they are all given in
     * /usr/include/linux/fs.h). Since we don't put
     * anything here, the system will keep the default
     * data, which in Unix is zeros (NULLs when taken as
     * pointers). */
  };



/* Inode operations for our proc file. We need it so
 * we'll have some place to specify the file operations
 * structure we want to use, and the function we use for
 * permissions. It's also possible to specify functions
 * to be called for anything else which could be done to
 * an inode (although we don't bother, we just put
 * NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* create */
    NULL, /* lookup */
    NULL, /* link */
    NULL, /* unlink */
    NULL, /* symlink */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* rename */
    NULL, /* readlink */
    NULL, /* follow_link */
    NULL, /* readpage */
    NULL, /* writepage */
    NULL, /* bmap */
    NULL, /* truncate */
```

```
    module_permission /* check for permissions */
  };


/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
    7, /* Length of the file name */
    "rw_test", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR,
    /* File mode - this is a regular file which
     * can be read by its owner, its group, and everybody
     * else. Also, its owner can write to it.
     *
     * Actually, this field is just for reference, it's
     * module_permission that does the actual check. It
     * could use this field, but in our implementation it
     * doesn't, for simplicity. */
    1,  /* Number of links (directories where the
         * file is referenced) */
    0, 0,  /* The uid and gid for the file -
            * we give it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File,
    /* A pointer to the inode structure for
     * the file, if we need it. In our case we
     * do, because we need a write function. */
    NULL
    /* The read function for the file. Irrelevant,
     * because we put it in the inode structure above */
  };


/* Module initialization and cleanup ****************** */

/* Initialize the module - register the proc file */
int init_module()

{
  /* Success if proc_register[_dynamic] is a success,
   * failure otherwise */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
   * inode number automatically if it is zero in the
   * structure, so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
```

```
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第5章　把设备文件用于输入

设备文件一般代表物理设备，而大多数物理设备既可用于输出，也可用于输入，所以Linux必须提供一些机制，以便内核中的设备驱动程序可以从进程获得输出信息，并把它发送到设备。要做到这一点，可以为输出打开设备文件，并且向它写信息，就像写普通的文件一样。在下面的例子中，这些任务是由 device_write完成的。

当然仅有上面这种方法是不够的。假设用户有一个与调制解调器相连接的串行口 (即使用户拥有的是一个内置式的调制解调器，从 CPU角度来看，它还是由一个与调制解调器相连的串行口来实现的，所以这样的用户也无需过多地苛求自己的想象力)。用户将会做的最自然的事情就是使用设备文件来把信息写到调制解调器 (调制解调器命令或者数据将会通过电话线来传送)，并且利用设备文件从调制解调器读信息 (命令的响应或者数据也是通过电话线接收的)。然而，这会带来一个很明显的问题：如果用户需要与串行口本身交换信息的话，用户该怎么办？例如用户可能发送有关数据发送和接收的速率的值。

在Unix中，可以使用一个称为 ioctl的特殊函数来解决这个问题 (ioctl是输入输出控制的英文缩写)。每个设备都有属于自己的 ioctl命令，可以是读 ioctl(从进程把信息发送到内核)、写 ioctl(把信息返回到进程)、都有或者都没有。调用 ioctl函数必须带上三个参数：适当的设备文件的文件描述符， ioctl编号以及另外一个长整型的参数，用户可以使用这个长整型参数来传送任何信息。

ioctl编号是由主设备编号、 ioctl类型、命令以及参数的类型这几者编码而成。这个 ioctl编号通常是由一个头文件中的宏调用 (取决于类型的不同，可以是 _IO、_IOR、_IOW或者 _IOWR)来创建的。然后，将要使用 ioctl的程序以及内核模块都必须通过 #include命令包含这个头文件。前者包含这个头文件是为了生成适当的 ioctl，而后者是为了能理解它。在下面的例子中，头文件的名称是 chardev.h，而使用它的程序是 ioctl.c。

如果用户希望在自己的内核模块中使用 ioctl，最好是接受正式的 ioctl的约定，这样如果偶尔得到别人的 ioctl，或者如果别人获得了你的 ioctl，你可以知道那些地方出现了错误。如果读者想知道更多的信息，可以查询 Documentation/ioctl -number.txt下的内核源代码树。

```
ex chardev.c

/* chardev.c
 *
 * Create an input/output character device
 */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */
```

```
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */

/* The character device definitions are here */
#include <linux/fs.h>

/* A wrapper which does next to nothing at
 * at present, but may help for compatibility
 * with future versions of Linux */
#include <linux/wrapper.h>
/* Our own ioctl numbers */
#include "chardev.h"


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif



#define SUCCESS 0


/* Device Declarations ******************************** */


/* The name for our device, as it will appear in
 * /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message for the device */
#define BUF_LEN 80
```

```
/* Is the device open right now? Used to prevent
 * concurent access into the same device */
static int Device_Open = 0;

/* The message the device will give when asked */
static char Message[BUF_LEN];

/* How far did the process reading the message get?
 * Useful if the message is larger than the size of the
 * buffer we get to fill in device_read. */
static char *Message_Ptr;
/* This function is called whenever a process attempts
 * to open the device file */
static int device_open(struct inode *inode,
                       struct file *file)
{
#ifdef DEBUG
  printk ("device_open(%p)\n", file);
#endif

  /* We don't want to talk to two processes at the
   * same time */
  if (Device_Open)
    return -EBUSY;

  /* If this was a process, we would have had to be
   * more careful here, because one process might have
   * checked Device_Open right before the other one
   * tried to increment it. However, we're in the
   * kernel, so we're protected against context switches.
   *
   * This is NOT the right attitude to take, because we
   * might be running on an SMP box, but we'll deal with
   * SMP in a later chapter.
   */

  Device_Open++;

  /* Initialize the message */
  Message_Ptr = Message;

  MOD_INC_USE_COUNT;

  return SUCCESS;
}


/* This function is called when a process closes the
 * device file. It doesn't have a return value because
 * it cannot fail. Regardless of what else happens, you
 * should always be able to close a device (in 2.0, a 2.2
```

```
 * device file could be impossible to close). */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                          struct file *file)
#else
static void device_release(struct inode *inode,
                          struct file *file)
#endif
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif

  /* We're now ready for our next caller */
  Device_Open --;

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
#endif
}



/* This function is called whenever a process which
 * has already opened the device file attempts to
 * read from it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(
    struct file *file,
    char *buffer, /* The buffer to fill with the data */
    size_t length,     /* The length of the buffer */
    loff_t *offset) /* offset to the file */
#else
static int device_read(
    struct inode *inode,
    struct file *file,
    char *buffer,   /* The buffer to fill with the data */
    int length)     /* The length of the buffer
                     * (mustn't write beyond that!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;

#ifdef DEBUG
  printk("device_read(%p,%p,%d)\n",
    file, buffer, length);
#endif
```

```
  /* If we're at the end of the message, return 0
   * (which signifies end of file) */
  if (*Message_Ptr == 0)
    return 0;

  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {

    /* Because the buffer is in the user data segment,
     * not the kernel data segment, assignment wouldn't
     * work. Instead, we have to use put_user which
     * copies data from the kernel data segment to the
     * user data segment. */
    put_user(*(Message_Ptr++), buffer++);
    length --;
    bytes_read ++;
  }

#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif

   /* Read functions are supposed to return the number
    * of bytes actually inserted into the buffer */
  return bytes_read;
}


/* This function is called when somebody tries to
 * write into our device file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
                            const char *buffer,
                            size_t length,
                            loff_t *offset)
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  int i;

#ifdef DEBUG
  printk ("device_write(%p,%s,%d)",
    file, buffer, length);
#endif

  for(i=0; i<length && i<BUF_LEN; i++)
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buffer+i);
#else
    Message[i] = get_user(buffer+i);
#endif

  Message_Ptr = Message;

  /* Again, return the number of input characters used */
  return i;
}


/* This function is called whenever a process tries to
 * do an ioctl on our device file. We get two extra
 * parameters (additional to the inode and file
 * structures, which all device functions get): the number
 * of the ioctl called and the parameter given to the
 * ioctl function.
 *
 * If the ioctl is write or read/write (meaning output
 * is returned to the calling process), the ioctl call
 * returns the output of this function.
 */
int device_ioctl(
    struct inode *inode,
    struct file *file,
    unsigned int ioctl_num,/* The number of the ioctl */
    unsigned long ioctl_param) /* The parameter to it */
{
  int i;
  char *temp;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  char ch;
#endif

  /* Switch according to the ioctl called */
  switch (ioctl_num) {
    case IOCTL_SET_MSG:
      /* Receive a pointer to a message (in user space)
       * and set that to be the device's message. */

      /* Get the parameter given to ioctl by the process */
      temp = (char *) ioctl_param;

      /* Find the length of the message */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, temp);
      for (i=0; ch && i<BUF_LEN; i++, temp++)
        get_user(ch, temp);
```

```
#else
      for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
    ;
#endif

      /* Don't reinvent the wheel - call device_write */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      device_write(file, (char *) ioctl_param, i, 0);
#else
      device_write(inode, file, (char *) ioctl_param, i);
#endif
      break;

    case IOCTL_GET_MSG:
      /* Give the current message to the calling
       * process - the parameter we got is a pointer,
       * fill it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      i = device_read(file, (char *) ioctl_param, 99, 0);
#else
      i = device_read(inode, file, (char *) ioctl_param,
                      99);
#endif
      /* Warning - we assume here the buffer length is
       * 100. If it's less than that we might overflow
       * the buffer, causing the process to core dump.
       *
       * The reason we only allow up to 99 characters is
       * that the NULL which terminates the string also
       * needs room. */

      /* Put a zero at the end of the buffer, so it
       * will be properly terminated */
      put_user('\0', (char *) ioctl_param+i);
      break;
    case IOCTL_GET_NTH_BYTE:
      /* This ioctl is both input (ioctl_param) and
       * output (the return value of this function) */
      return Message[ioctl_param];
      break;
  }

  return SUCCESS;
}


/* Module Declarations ************************** */


/* This structure will hold the functions to be called
```

```
 * when a process does something to the device we
 * created. Since a pointer to this structure is kept in
 * the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */
struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  device_ioctl,   /* ioctl */
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,  /* flush */
#endif
  device_release  /* a.k.a. close */
};


/* Initialize the module - Register the character device */
int init_module()
{
  int ret_val;

  /* Register the character device (atleast try) */
  ret_val = module_register_chrdev(MAJOR_NUM,
                                   DEVICE_NAME,
                                   &Fops);

  /* Negative values signify an error */
  if (ret_val < 0) {
    printk ("%s failed with %d\n",
            "Sorry, registering the character device ",
            ret_val);
    return ret_val;
  }

  printk ("%s The major device number is %d.\n",
          "Registeration is a success",
          MAJOR_NUM);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod %s c %d 0\n", DEVICE_FILE_NAME,
          MAJOR_NUM);
  printk ("The device file name is important, because\n");
  printk ("the ioctl program assumes that's the\n");
  printk ("file you'll use.\n");

  return 0;
```

```
}


/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;

  /* Unregister the device */
  ret = module_unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
ex chardev.h

/* chardev.h - the header file with the ioctl definitions.
 *
 * The declarations here have to be in a header file,
 * because they need to be known both to the kernel
 * module (in chardev.c) and the process calling ioctl
 * (ioctl.c)
 */
#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>



/* The major device number. We can't rely on dynamic
 * registration any more, because ioctls need to know
 * it. */
#define MAJOR_NUM 100


/* Set the message of the device driver */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/* _IOR means that we're creating an ioctl command
 * number for passing information from a user process
 * to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device
 * number we're using.
 *
 * The second argument is the number of the command
 * (there could be several with different meanings).
 *
 * The third argument is the type we want to get from
```

```
 * the process to the kernel.
 */

/* Get the message of the device driver */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
 /* This IOCTL is used for output, to get the message
  * of the device driver. However, we still need the
  * buffer to place the message in to be input,
  * as it is allocated by the process.
  */


/* Get the n'th byte of the message */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
 /* The IOCTL is used for both input and output. It
  * receives from the user a number, n, and returns
  * Message[n]. */
/* The name of the device file */
#define DEVICE_FILE_NAME "char_dev"


#endif
ex ioctl.c

/* ioctl.c - the process to use ioctl's to control the
 * kernel module
 *
 * Until now we could have used cat for input and
 * output. But now we need to do ioctl's, which require
 * writing our own process.
 */

/* Copyright (C) 1998 by Ori Pomerantz */


/* device specifics, such as ioctl numbers and the
 * major device file. */
#include "chardev.h"


#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */


/* Functions for the ioctl calls */

ioctl_set_msg(int file_desc, char *message)
{
  int ret_val;
```

```
  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

  if (ret_val < 0) {
    printf ("ioctl_set_msg failed:%d\n", ret_val);
    exit(-1);
  }
}
ioctl_get_msg(int file_desc)
{
  int ret_val;
  char message[100];

  /* Warning - this is dangerous because we don't tell
   * the kernel how far it's allowed to write, so it
   * might overflow the buffer. In a real production
   * program, we would have used two ioctls - one to tell
   * the kernel the buffer length and another to give
   * it the buffer to fill
   */
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

  if (ret_val < 0) {
    printf ("ioctl_get_msg failed:%d\n", ret_val);
    exit(-1);
  }

  printf("get_msg message:%s\n", message);
}



ioctl_get_nth_byte(int file_desc)
{
  int i;
  char c;

  printf("get_nth_byte message:");

  i = 0;
  while (c != 0) {
    c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

    if (c < 0) {
      printf(
      "ioctl_get_nth_byte failed at the %d'th byte:\n", i);
      exit(-1);
    }

    putchar(c);
  }
  putchar('\n');
}
```

```
/* Main - Call the ioctl functions */
main()
{
  int file_desc, ret_val;
  char *msg = "Message passed by ioctl\n";

  file_desc = open(DEVICE_FILE_NAME, 0);
  if (file_desc < 0) {
    printf ("Can't open device file: %s\n",
            DEVICE_FILE_NAME);
    exit(-1);
  }

  ioctl_get_nth_byte(file_desc);
  ioctl_get_msg(file_desc);
  ioctl_set_msg(file_desc, msg);

  close(file_desc);
}
```

第6章 启动参数

在前面所给出的许多例子中，我们不得不把一些东西硬塞进内核模块中，如 /proc文件的文件名或者设备的主设备编号，这样我们就可以使用该设备的 ioctl命令。但这是与 Unix和Linux的宗旨背道而驰的，Unix和Linux提倡编写用户所习惯的易于使用的程序。

在程序或者内核模块开始工作之前，如果希望告诉它一些它所需要的信息，可以使用命令行参数。如果是内核模块，我们不需要使用 argc和argv —— 相反，我们还有更好的选择。我们可以在内核模块中定义一些全局变量，然后使用 insmod命令，它将替我们给这些变量赋值。

在下面这个内核模块中，我们定义了两个全局变量： str1和str2。用户所需做的全部工作就是编译该内核模块，然后运行 insmod str1=xxx str2=yyy。当调用init_module时，str1将指向字符串“ xxx ”，而str2将指向字符串“ yyy ”。

在版本2中，对这些参数不进行类型检查。如果 str1或者str2的第一个字符是一个数字，则内核将用该整数的值填充这个变量，而不会用指向字符串的指针去填充它。如果用户对此不太确定，那么就必须亲自去检查一下。

而另一方面，在版本2.2中，用户使用宏MACRO_PARM告诉insmod自己希望参数、它的名称以及类型是什么样的。这就解决了类型问题，并且允许内核模块接受那些以数字开头的字符串。

```
ex param.c
/* param.c
 *
 * Receive command line parameters at module installation
 */

/* Copyright (C) 1998-99 by Ori Pomerantz */




/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


#include <stdio.h>  /* I need NULL */
```

```
/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Emmanuel Papirakis:
 *
 * Parameter names are now (2.2) handled in a macro.
 * The kernel doesn't resolve the symbol names
 * like it seems to have once did.
 *
 * To pass parameters to a module, you have to use a macro
 * defined in include/linux/modules.h (line 176).
 * The macro takes two parameters. The parameter's name and
 * it's type. The type is a letter in double quotes.
 * be a string.
 */


char *str1, *str2;


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(str1, "s");
MODULE_PARM(str2, "s");
#endif


/* Initialize the module - show the parameters */
int init_module()
{
  if (str1 == NULL || str2 == NULL) {
    printk("Next time, do insmod param str1=<something>");
    printk("str2=<something>\n");
  } else
    printk("Strings:%s and %s\n", str1, str2);

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  printk("If you try to insmod this module twice,");
  printk("(without rmmod'ing\n");
  printk("it first), you might get the wrong");
  printk("error message:\n");
  printk("'symbol for parameters str1 not found'.\n");
#endif

  return 0;
}
```

```
}


/* Cleanup */
void cleanup_module()
{
}
```

第7章 系统调用

到现在为止，我们所做过的唯一的工作就是使用一个定义好的内核机制来注册 /proc文件和设备处理程序。如果用户仅仅希望做一个内核程序员份内的工作，例如编写设备驱动程序，那么以前我们所学的知识已经足够了。但是如果用户想做一些不平凡的事，比如在某些方面，在某种程度上改变一下系统的行为，那应该怎么办呢？答案是，几乎全部要靠自己。

这就是内核编程之所以危险的原因。在编写下面的例题时，我关掉了系统调用 open。这将意味着我不能打开任何文件，不能运行任何程序，甚至不能关闭计算机。我只能把电源开关拔掉。幸运的是，我没有删除掉任何文件。为了保证自己也不丢失任何文件，请读者在执行insmod和rmmod之前先运行sync。

现在让我们忘记/proc文件，忘记设备文件，他们只不过是无关大雅的细节问题。实现内核通信机制的“真正”进程是系统调用，它是被所有进程所使用的进程。当某进程向内核请求服务时(例如打开文件、产生一个新进程、或者请求更多的内存)，它所使用的机制就是系统调用。如果用户希望以一种有趣的方式改变内核的行为，也需要依靠系统调用。顺便说一下，如果用户想知道程序使用的是哪个系统调用，可以运行命令 strace <command> <arguments>。

一般来说，进程是不能访问内核的。它不能访问内核存储，它也不能调用内核函数。CPU的硬件保证了这一点(那就是为什么称之为“保护模式”的原因)。系统调用是这条通用规则的一个特例。在进行系统调用时，进程以适当的值填充注册程序，然后调用一条特殊的指令，而这条指令是跳转到以前定义好的内核中的某个位置 (当然，用户进程可以读那个位置，但却不能对它进行写的操作)。在Intel CPU下，以上任务是通过中断0x80来完成的。硬件知道一旦跳转到这个位置，用户的进程就不再是在受限制的用户模式下运行了，而是作为操作系统内核来运行——于是用户就被允许干所有他想干的事。

进程可以跳转到的那个内核中的位置称为 system_call。那个位置上的过程检查系统调用编号(系统调用编号可以告诉内核进程所请求的是什么服务)。然后，该过程查看系统调用表(sys_call_table)，找出想要调用的内核函数的地址，然后调用那个函数。在函数返回之后，该过程还要做一些系统检查工作，然后再返回到进程 (或者如果进程时间已用完，则返回到另一个进程)。如果读者想读这段代码，可以查看源文件 arch/<architecture>/kernel/entry.S，它就在ENTRY(system_call)那一行的后面。

这样看来，如果我们想要改变某个系统调用的工作方式，我们需要编写自己的函数来实现它(通常是加入一点自己的代码，然后再调用原来的函数)，然后改变sys_call_table表中的指针，使它指向我们的函数。因为我们的函数将来可能会被删除掉，而我们不想使系统处于一个不稳定的状态，所以必须用 cleanup_module使sys_call_table表恢复到它的原始状态，这是很重要的。

本章的源代码就是这样一个内核模块的例子。我们想要“侦听”某个特定的用户。无论何时，只要那个用户一打开某个文件，程序就会用 printk打印出一个消息。为了做到这一点，我们用自己的函数代替了用来打开文件的那个系统调用。我们的函数称为 our_sys_open。该函

数查看当前进程的uid(用户的ID)，如果它就是我们想要侦听的 uid，它就调用printk，显示出将要打开的文件的名称。接下来，不管当前进程的 uid是不是想要侦听的uid，该函数都使用同样的参数调用原来的open函数，真正地打开那个文件。

init_module函数替换sys_call_table表中相应的位置，并把原来的指针存放在一个变量中；而cleanup_module函数则使用那个变量把一切都恢复成原来正常的状态。这种方法是具有一定的危险性的，因为可能有两个内核模块都修改同一个系统调用。现在假设有两个内核模块A和B。A模块打开文件的系统调用是 A_open，而B的系统调用是B_open。当把A插入到内核中时，系统调用被换成了A_open，它在被调用时将会调用原来的sys_open。接下来，当B被插入到内核中时，系统调用将被替换成 B_open，它在被调用时，将会调用它自以为是原始系统调用的那个系统调用，即A_open。

现在假设B先被删除，那么一切都将正常 —— 系统调用将被恢复成A_open，而A_open会调用原始的系统调用。然而，如果 A先被删除然后B被删除，系统将会崩溃。删除 A时系统调用被恢复成原始的sys_open, B_open将被忽略。接着，当删除 B时，B将把系统调用恢复成它自认为是原始的系统调用，即 A_open，而A_open已经不再位于内存中了。乍一看上去，好象用户可以检查系统调用是不是打开文件函数，如果是就不做任何修改 (这样在删除B时它就不会改变系统调用)，似乎这样可以避免问题的发生。但这样做将会导致一个更为严重的问题。当A被删除时，它看到系统调用已经被改变为 B_open，而不再指向A_open，所以A在从内存中删除前不会将系统调用恢复为 sys_open。不幸的是， B_open仍将试图调用 A_open，而A_open已不在内存中了，这样，甚至还不到删除 B时系统就将崩溃。

我认为有两种方法可以解决这个问题。第一个方法是把系统调用恢复为原始的值：sys_open。不幸的是，sys_open不是/proc/ksyms中内核系统表的一部分，所以我们不能访问它。另一个方法是一旦装入了模块，马上设立一个引用计数器，以防止根用户把它 rmmod掉。对于产品模块来说，这样做是很好的，但却不适合于作为教学的例子 —— 这就是我没有在这里实现它的原因。

```
ex syscall.c

/* syscall.c
 *
 * System call "stealing" sample
 */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
```

```
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h>  /* The list of system calls */

/* For the current (process) structure, we need
 * this to know who the current user is. */
#include <linux/sched.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#endif



/* The system call table (a table of functions). We
 * just define this as external, and the kernel will
 * fill it up for us when we are insmod'ed
 */
extern void *sys_call_table[];


/* UID we want to spy on - will be filled from the
 * command line */
int uid;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(uid, "i");
#endif

/* A pointer to the original system call. The reason
 * we keep this, rather than call the original function
 * (sys_open), is because somebody else might have
 * replaced the system call before us. Note that this
 * is not 100% safe, because if another module
 * replaced sys_open before us, then when we're inserted
 * we'll call the function in that module--and it
 * might be removed before we are.
 *
 * Another reason for this is that we can't get sys_open.
 * It's a static variable, so it is not exported. */
```

```
asmlinkage int (*original_call)(const char *, int, int);


/* For some reason, in 2.2.3 current->uid gave me
 * zero, not the real user ID. I tried to find what went
 * wrong, but I couldn't do it in a short time, and
 * I'm lazy - so I'll just use the system call to get the
 * uid, the way a process would.
 *
 * For some reason, after I recompiled the kernel this
 * problem went away.
 */
asmlinkage int (*getuid_call)();
/* The function we'll replace sys_open (the function
 * called when you call the open system call) with. To
 * find the exact prototype, with the number and type
 * of arguments, we find the original function first
 * (it's at fs/open.c).
 *
 * In theory, this means that we're tied to the
 * current version of the kernel. In practice, the
 * system calls almost never change (it would wreck havoc
 * and require programs to be recompiled, since the system
 * calls are the interface between the kernel and the
 * processes).
 */
asmlinkage int our_sys_open(const char *filename,
                            int flags,
                            int mode)
{
  int i = 0;
  char ch;

  /* Check if this is the user we're spying on */
  if (uid == getuid_call()) {
   /* getuid_call is the getuid system call,
    * which gives the uid of the user who
    * ran the process which called the system
    * call we got */

    /* Report the file, if relevant */
    printk("Opened file by %d: ", uid);
    do {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, filename+i);
#else
      ch = get_user(filename+i);
#endif
      i++;
      printk("%c", ch);
    } while (ch != 0);
```

```
      printk("\n");
    }

    /* Call the original sys_open - otherwise, we lose
     * the ability to open files */
    return original_call(filename, flags, mode);
}
/* Initialize the module - replace the system call */
int init_module()
{
    /* Warning - too late for it now, but maybe for
     * next time... */
    printk("I'm dangerous. I hope you did a ");
    printk("sync before you insmod'ed me.\n");
    printk("My counterpart, cleanup_module(), is even");
    printk("more dangerous. If\n");
    printk("you value your file system, it will ");
    printk("be \"sync; rmmod\" \n");
    printk("when you remove this module.\n");

    /* Keep a pointer to the original function in
     * original_call, and then replace the system call
     * in the system call table with our_sys_open */
    original_call = sys_call_table[__NR_open];
    sys_call_table[__NR_open] = our_sys_open;

    /* To get the address of the function for system
     * call foo, go to sys_call_table[__NR_foo]. */

    printk("Spying on UID:%d\n", uid);

    /* Get the system call for getuid */
    getuid_call = sys_call_table[__NR_getuid];

    return 0;
}


/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
    /* Return the system call back to normal */
    if (sys_call_table[__NR_open] != our_sys_open) {
        printk("Somebody else also played with the ");
        printk("open system call\n");
        printk("The system may be left in ");
        printk("an unstable state.\n");
    }

    sys_call_table[__NR_open] = original_call;
}
```

第8章 阻塞处理

如果有人想让你做一些目前无法做到的事，你会怎么处理呢？如果打扰你的是某个人的话，你可以说的唯一的一句话是：“现在不行，我很忙，你走吧！”但是如果是进程让内核模块做一些目前它无法处理的事，内核模块却可以有另一种处理方法。内核可以让进程睡眠，直到能够为它服务为止。毕竟，随时都会有进程被内核置为睡眠状态或者唤醒 (这就是多个进程看上去好象同时在一个CPU上运行的道理)。

本章的内核模块就是这样的一个例子。该文件 (名为/proc/sleep)每次只能被一个进程所打开。如果文件早已经被打开，内核模块将调用 module_interruptible_sleep_on。这个函数把任务的状态改为TASK_INTERRUPTIBLE(任务是内核数据结构，它包含着有关它所处的进程和系统调用的信息，如果存在系统调用的话)，把它加入到WaitQ当中，这就意味着任务在被唤醒之前将不会运行。WaitQ是等待访问文件的任务队列。然后，函数调用上下文调度程度，切换到另一个拥有CPU时间的进程。

在进程结束了文件操作以后，进程将关闭文件，并调用 module_close。该函数将唤醒队列中的所有进程(没有只唤醒一个进程的机制)，然后函数返回，刚刚关闭文件的那个进程就可以继续运行了。如果那个进程的时间片用完了，则调度程度将会及时判断出这一点，并将 CPU的控制权交给另一个进程。最后，等待队列中的某个进程将会被调度程序授予 CPU的控制权。该进程将从紧接着调用 module_interruptible_sleep_on后面的那个点开始执行。然后它会设置一个全局变量，告诉其它所有进程文件依然打开着，然后该进程将继续执行。当其它进程获得CPU时间片时，它们将看到那个全局变量，于是继续睡眠。

更为有趣的是，并不是只有 module_close才能唤醒那些等待访问文件的进程。一个信号，例如Ctrl+C(SIGINT)也可以唤醒进程。在那种情况下，我们希望立刻用 EINTR返回。这是很重要的，只有这样用户才能在进程接收到文件之前杀死那个进程。

还需要记住一点，有时候进程并不想睡眠；它们希望或者立刻拿到它们想要的东西，或者直接告诉它们这是不可能的。这样的进程在打开文件时使用标志 O_NONBLOCK。内核在进行该操作时，将会返回错误代码 EAGAIN来作响应，否则该操作就将阻塞，就像下面例子中的打开文件操作一样。本章的源目录中有一个程序 cat_noblock，可以用于带O_NONBLOCK标志打开一个文件。

```
ex sleep.c

/* sleep.c - create a /proc file, and if several
 * processes try to open it at the same time, put all
 * but one to sleep */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */
```

```
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary because we use proc fs */
#include <linux/proc_fs.h>

/* For putting processes to sleep and waking them up */
#include <linux/sched.h>
#include <linux/wrapper.h>
/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif


/* The module's file functions ********************** */


/* Here we keep the last message received, to prove
 * that we can process our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];


/* Since we use the file operations struct, we can't use
 * the special proc output provisions - we have to use
 * a standard read function, which is this function */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,    /* The file read */
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    size_t len,  /* The length of the buffer */
    loff_t *offset) /* Offset in the file - ignore */
#else
static int module_output(
    struct inode *inode, /* The inode read */
```

```
      struct file *file,   /* The file read */
      char *buf, /* The buffer to put data to (in the
                  * user segment) */
      int len)  /* The length of the buffer */
  #endif
  {
    static int finished = 0;
    int i;
    char message[MESSAGE_LENGTH+30];
  /* Return 0 to signify end of file - that we have
   * nothing more to say at this point. */
   if (finished) {
     finished = 0;
     return 0;
   }

   /* If you don't understand this by now, you're
    * hopeless as a kernel  programmer. */
   sprintf(message, "Last input:%s\n", Message);
   for(i=0; i<len && message[i]; i++)
     put_user(message[i], buf+i);

   finished = 1;
   return i;  /* Return the number of bytes "read" */
}


/* This function receives input from the user when
 * the user writes to the /proc file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
    struct file *file,    /* The file itself */
    const char *buf,      /* The buffer with input */
    size_t length,        /* The buffer's length */
    loff_t *offset)       /* offset to file - ignore */
#else
static int module_input(
    struct inode *inode, /* The file's inode */
    struct file *file,   /* The file itself */
    const char *buf,     /* The buffer with the input */
    int length)          /* The buffer's length */
#endif
{
  int i;

  /* Put the input into Message, where module_output
   * will later be able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buf+i);
#else
```

```
    Message[i] = get_user(buf+i);
#endif
/* we want a standard, zero terminated string */
    Message[i] = '\0';
  /* We need to return the number of input
   * characters used */
  return i;
}

/* 1 if the file is currently open by somebody */
int Already_Open = 0;

/* Queue of processes who want our file */
static struct wait_queue *WaitQ = NULL;


/* Called when the /proc file is opened */
static int module_open(struct inode *inode,
                       struct file *file)
{
  /* If the file's flags include O_NONBLOCK, it means
   * the process doesn't want to wait for the file.
   * In this case, if the file is already open, we
   * should fail with -EAGAIN, meaning "you'll have to
   * try again", instead of blocking a process which
   * would rather stay awake. */
  if ((file->f_flags & O_NONBLOCK) && Already_Open)
    return -EAGAIN;

  /* This is the correct place for MOD_INC_USE_COUNT
   * because if a process is in the loop, which is
   * within the kernel module, the kernel module must
   * not be removed. */
  MOD_INC_USE_COUNT;

  /* If the file is already open, wait until it isn't */
  while (Already_Open)
  {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    int i, is_sig=0;
#endif

    /* This function puts the current process,
     * including any system calls, such as us, to sleep.
     * Execution will be resumed right after the function
     * call, either because somebody called
     * wake_up(&WaitQ) (only module_close does that,
     * when the file is closed) or when a signal, such
     * as Ctrl-C, is sent to the process */
    module_interruptible_sleep_on(&WaitQ);
```

```
    /* If we woke up because we got a signal we're not
     * blocking, return  -EINTR (fail the system call).
     * This allows processes to be killed or stopped. */


/*
 * Emmanuel Papirakis:
 *
 * This is a little update to work with 2.2.*. Signals
 * now are contained in two words (64 bits) and are
 * stored in a structure that contains an array of two
 * unsigned longs. We now have to make 2 checks in our if.
 *
 * Ori Pomerantz:
 *
 * Nobody promised me they'll never use more than 64
 * bits, or that this book won't be used for a version
 * of Linux with a word size of 16 bits. This code
 * would work in any case.
 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)

    for(i=0; i<_NSIG_WORDS && !is_sig; i++)
      is_sig = current->signal.sig[i] &
        ~current->blocked.sig[i];
    if (is_sig) {
#else
    if (current->signal & ~current->blocked) {
#endif
      /* It's important to put MOD_DEC_USE_COUNT here,
       * because for processes where the open is
       * interrupted there will never be a corresponding
       * close. If we don't decrement the usage count
       * here, we will be left with a positive usage
       * count which we'll have no way to bring down to
       * zero, giving us an immortal module, which can
       * only be killed by rebooting the machine. */
      MOD_DEC_USE_COUNT;
      return -EINTR;
    }
  }

  /* If we got here, Already_Open must be zero */
  /* Open the file */
  Already_Open = 1;
  return 0;  /* Allow the access */
}

/* Called when the /proc file is closed */
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  /* Set Already_Open to zero, so one of the processes
   * in the WaitQ will be able to set Already_Open back
   * to one and to open the file. All the other processes
   * will be called when Already_Open is back to one, so
   * they'll go back to sleep. */
  Already_Open = 0;

  /* Wake up all the processes in WaitQ, so if anybody
   * is waiting for the file, they can have it. */
  module_wake_up(&WaitQ);

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* success */
#endif
}



/* This function decides whether to allow an operation
 * (return zero) or not allow it (return a non-zero
 * which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file
 * permissions. The permissions returned by ls -l are
 * for reference only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but
   * only root (uid 0) may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /* If it's anything else, access is denied */
  return -EACCES;
}
```

```
/* Structures to register as the /proc file, with
 * pointers to all the relevant functions. ********** */


/* File operations for our proc file. This is where
 * we place pointers to all the functions called when
 * somebody tries to do something to our file. NULL
 * means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,/* called when the /proc file is opened */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* flush */
#endif
    module_close      /* called when it's classed */
  };



/* Inode operations for our proc file. We need it so
 * we'll have somewhere to specify the file operations
 * structure we want to use, and the function we use for
 * permissions. It's also possible to specify functions
 * to be called for anything else which could be done to an
 * inode (although we don't bother, we just put NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* create */
    NULL, /* lookup */
    NULL, /* link */
    NULL, /* unlink */
    NULL, /* symlink */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* rename */
    NULL, /* readlink */
    NULL, /* follow_link */
    NULL, /* readpage */
    NULL, /* writepage */
    NULL, /* bmap */
    NULL, /* truncate */
```

```
    module_permission /* check for permissions */
  };

/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
    5, /* Length of the file name */
    "sleep", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR,
    /* File mode - this is a regular file which
     * can be read by its owner, its group, and everybody
     * else. Also, its owner can write to it.
     *
     * Actually, this field is just for reference, it's
     * module_permission that does the actual check. It
     * could use this field, but in our implementation it
     * doesn't, for simplicity. */
    1,  /* Number of links (directories where the
         * file is referenced) */
    0, 0,  /* The uid and gid for the file - we give
            * it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File,
    /* A pointer to the inode structure for
     * the file, if we need it. In our case we
     * do, because we need a write function. */
    NULL  /* The read function for the file.
           * Irrelevant, because we put it
           * in the inode structure above */
  };

/* Module initialization and cleanup **************** */


/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register_dynamic is a success,
   * failure otherwise */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* proc_root is the root directory for the proc
   * fs (/proc). This is where we want our file to be
   * located.
   */
```

```
}


/* Cleanup - unregister our file from /proc. This could
 * get dangerous if there are still processes waiting in
 * WaitQ, because they are inside our open function,
 * which will get unloaded. I'll explain how to avoid
 * removal of a kernel module in such a case in
 * chapter 10. */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第9章　替换printk

在第1章，我曾经提到过X编程和内核模块编程不能混为一谈。这句话在开发内核模块时是正确的。但是在实际应用中，用户可能希望向运行 tty命令的那个模块发送消息。在释放内核模块以后，这对于识别错误是非常重要的。因为该内核模块将会被所有使用 tty命令的模块所使用。

通过使用current可以做到这一点， current是一个指向当前正在运行的任务的指针，通过使用它可以获得当前任务的tty结构。然后查看tty结构，可以找到指向写字符串的函数的指针，我们就是用这个指针向tty写字符串的。

```
ex printk.c

/* printk.c - send textual output to the tty you're
 * running on, regardless of whether it's passed
 * through X11, telnet, etc. */


/* Copyright (C) 1998 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work
*/
#include <linux/module.h>   /* Specifically, a module */
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary here */
#include <linux/sched.h>    /* For current */
#include <linux/tty.h>      /* For the tty declarations */


/* Print the string to the appropriate tty, the one
 * the current task uses */
void print_string(char *str)
{
  struct tty_struct *my_tty;

  /* The tty for the current task */
  my_tty = current->tty;
```

```
/* If my_tty is NULL, it means that the current task
 * has no tty you can print to (this is possible, for
 * example, if it's a daemon). In this case, there's
 * nothing we can do. */
if (my_tty != NULL) {

  /* my_tty->driver is a struct which holds the tty's
   * functions, one of which (write) is used to
   * write strings to the tty. It can be used to take
   * a string either from the user's memory segment
   * or the kernel's memory segment.
   *
   * The function's first parameter is the tty to
   * write to, because the  same function would
   * normally be used for all tty's of a certain type.
   * The second parameter controls whether the
   * function receives a string from kernel memory
   * (false, 0) or from user memory (true, non zero).
   * The third parameter is a pointer to a string,
   * and the fourth parameter is the length of
   * the string.
   */
  (*(my_tty->driver).write)(
      my_tty, /* The tty itself */
      0, /* We don't take the string from user space */
  str, /* String */
  strlen(str));  /* Length */

  /* ttys were originally hardware devices, which
   * (usually) adhered strictly to the ASCII standard.
   * According to ASCII, to move to a new line you
   * need two characters, a carriage return and a
   * line feed. In Unix, on the other hand, the
   * ASCII line feed is used for both purposes - so
   * we can't just use \n, because it wouldn't have
   * a carriage return and the next line will
   * start at the column right
   *                          after the line feed.
   *
   * BTW, this is the reason why the text file
   *  is different between Unix and Windows.
   * In CP/M and its derivatives, such as MS-DOS and
   * Windows, the ASCII standard was strictly
   * adhered to, and therefore a new line requires
   * both a line feed and a carriage return.
   */
  (*(my_tty->driver).write)(
    my_tty,
    0,
    "\015\012",
    2);
```

```
  }
}


/* Module initialization and cleanup ****************** */


/* Initialize the module - register the proc file */
int init_module()
{
  print_string("Module Inserted");

  return 0;
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  print_string("Module Removed");
}
```

第10章 任务调度

常常有一些“内务处理”任务需要我们定时或者经常去做。如果任务是由进程去完成的，我们可以把它放在crontab文件中。如果任务要由内核模块来完成，那么我们将面临两种选择。第一种方案是把一个进程放在 crontab文件中，该进程在需要的时候通过系统调用唤醒模块。例如，通过打开一个文件来做到这一点。第三种方案需要在 crontab中运行一个新进程，把一个新的可执行程序读入内存，而这一切都只是为了唤醒一个早已经位于内存中的内核模块这样做效率是非常低的。

除了以上这两种方法以外，我们还可以创建一个函数，在每次时钟中断时调用一次那个函数。为了做到这一点，需要创建一个任务，存放在结构 tq_struct中，而该结构将存放指向函数的指针。接着，我们使用 queue_task把那个任务放置到一个称为 tq_timer的任务列表中，该任务列表中的任务都将在下次时钟中断时执行。由于我们希望该函数能继续执行下去，无论该函数何时被调用，我们一定要把它放回到 tq_timer中，这样在下一次时钟中断时它还可以执行。

在这里还需要记住一点。当使用 rmmod命令删除一个模块时，首先需要检查它的引用计数器值。如果计数器的值为 0，则调用module_cleanup。然后，该模块以及它的所有函数就被从内存中删除掉了。没有人会记得检查一下看看时钟的任务列表中是否碰巧包含了一个指向这些函数中某个函数的指针，而该函数已经不再是可用的了。很长一段时间以后 (这是从计算机的角度来说的，从人的角度来看这段时间不值一提，有可能比百分之一秒还短)，内核发生了一次时钟中断，并试图调用任务列表中的函数。不幸的是，该函数早已经不在那里了。在大多数情况下，该函数原来所在的内存页还没有被使用，这时用户得到的将是一段难看的错误信息。但如果那个内存位置已经存放了一些新的其它的代码，事情就变得非常糟糕了。不幸的是，我们还没有一种简便的方法可以从任务列表中取消一个任务的注册。

由于cleanup_module不能返回错误代码(它是一个void函数)，解决的方法是根本不让它返回，相反，它调用 sleep_on或者module_sleep_on，把rmmod进程置为睡眠状态。而在此之前，它会设置一个全局变量，通知那些将要在时钟中断时调用的函数停止连接。然后，在下一次时钟中断发生时， rmmod进程将被唤醒，我们的函数已经不在队列中了，这时就可以安全地删除模块。

```
ex sched.c

/* sched.c - scheduale a function to be called on
 * every timer interrupt. */


/* Copyright (C) 1998 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
```

```
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>

/* We scheduale tasks here */
#include <linux/tqueue.h>

/* We also need the ability to put ourselves to sleep
 * and wake up later */
#include <linux/sched.h>

/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
/* The number of times the timer interrupt has been
 * called so far */
static int TimerIntrpt = 0;


/* This is used by cleanup, to prevent the module from
 * being unloaded while intrpt_routine is still in
 * the task queue */
static struct wait_queue *WaitQ = NULL;

static void intrpt_routine(void *);


/* The task queue structure for this task, from tqueue.h */
static struct tq_struct Task = {
  NULL,   /* Next item in list - queue_task will do
           * this for us */
  0,      /* A flag meaning we haven't been inserted
           * into a task queue yet */
  intrpt_routine, /* The function to run */
  NULL    /* The void* parameter for that function */
};

/* This function will be called on every timer
 * interrupt. Notice the void* pointer - task functions
```

```
 * interrupt. Notice the void* pointer - task functions
 * can be used for more than one purpose, each time
 * getting a different parameter. */
static void intrpt_routine(void *irrelevant)
{
  /* Increment the counter */
  TimerIntrpt++;

  /* If cleanup wants us to die */
  if (WaitQ != NULL)
    wake_up(&WaitQ);   /* Now cleanup_module can return */
  else
    /* Put ourselves back in the task queue */
    queue_task(&Task, &tq_timer);
}
/* Put data into the proc fs file. */
int procfile_read(char *buffer,
                  char **buffer_location, off_t offset,
                  int buffer_length, int zero)
{
  int len;  /* The number of bytes actually used */

  /* This is static so it will still be in memory
   * when we leave this function */
  static char my_buffer[80];

  static int count = 1;

  /* We give all of our information in one go, so if
   * the anybody asks us if we have more information
   * the answer should always be no.
   */
  if (offset > 0)
    return 0;

  /* Fill the buffer and get its length */
  len = sprintf(my_buffer,
                "Timer was called %d times so far\n",
                TimerIntrpt);
  count++;

  /* Tell the function which called us where the
   * buffer is */
  *buffer_location = my_buffer;

  /* Return the length */
  return len;
}


struct proc_dir_entry Our_Proc_File =
```

```
{
  0, /* Inode number - ignore, it will be filled by
      * proc_register_dynamic */
  5, /* Length of the file name */
  "sched", /* The file name */
  S_IFREG | S_IRUGO,
  /* File mode - this is a regular file which can
   * be read by its owner, its group, and everybody
   * else */
  1,  /* Number of links (directories where
         * the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give
            * it to root */
    80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the
           * inode (linking, removing, etc.) - we don't
           * support any. */
    procfile_read,
    /* The read function for this file, the function called
     * when somebody tries to read something from it. */
    NULL
    /* We could have here a function to fill the
     * file's inode, to enable us to play with
     * permissions, ownership, etc. */
  };


/* Initialize the module - register the proc file */
int init_module()
{
  /* Put the task in the tq_timer task queue, so it
   * will be executed at next timer interrupt */
  queue_task(&Task, &tq_timer);

  /* Success if proc_register_dynamic is a success,
   * failure otherwise */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup */
void cleanup_module()
{
  /* Unregister our /proc file */
  proc_unregister(&proc_root, Our_Proc_File.low_ino);

  /* Sleep until intrpt_routine is called one last
```

```
 * time. This is necessary, because otherwise we'll
 * deallocate the memory holding intrpt_routine and
 * Task while tq_timer still references them.
 * Notice that here we don't allow signals to
 * interrupt us.
    *
    * Since WaitQ is now not NULL, this automatically
    * tells the interrupt routine it's time to die. */
  sleep_on(&WaitQ);
}
```

第11章　中断处理程序

除了第10章以外，到目前为止我们在内核中所做的所有工作都是为了响应进程的请求，或者是处理一个特殊的文件，发送 ioctl，或者是发出一个系统调用。但是内核的任务并不仅仅是为了响应进程请求。另一个任务也是同样重要的，那就是内核还需要和与机器相连接的硬件进行通信。

在CPU和计算机的其它硬件之间有两种相互交互的方式。第一种是 CPU向硬件发出命令，另一种是硬件需要告诉 CPU一些事情。第二种方式称之为中断，相比较而言，中断要难实现得多，这是因为它需要在硬件方便的时候进行处理，而不是在 CPU方便的时候去处理。一般来说，每个硬件设备都会有一个数量相当少的 RAM，如果在可以读它们的信息的时候用户没有去读，则这些信息将丢失。

在Linux下，硬件中断称为IRQ(即Interrupt Request的简称，中断请求)。IRQ分为两种类型：短的和长的，短IRQ是指需要的时间周期非常短，在这段时间内，机器的其它部分将被阻塞，并且不再处理其它的中断。而长 IRQ是指需要的时间相对长一些，在这段时间内也可能会发生别的中断(但是同一个设备上不会再产生中断)。如果有可能的话，中断处理程序还是处理长IRQ要好一些。

当CPU接收到一个中断时，它将停下手中所有的工作 (除非它正在处理一个更为重要的中断，在那种情况下，只有处理完那个更重要的中断以后， CPU才会去处理这个中断)，在栈中保存某些特定的参数，并调用中断处理程序。这就意味着在中断处理程序内部有些特定的事情是不允许的，因为系统处于一个未知的状态下。为了解决这个问题，中断处理程序应该做那些需要立刻去做的事情，通常是从硬件读一些信息或者向硬件发送一些信息，在稍后的某时刻再调度去做新信息的处理工作 (这称为“ 底半处理 ”)并返回。内核必须保证尽快调用底半处理 —— 在进行底半处理工作时，内核模块中允许做的所有工作都可以做。

要实现这一点，可以调用 request_irq，这样一来，在接收到相关 IRQ时，就可以调用用户的中断处理程序 (在Intel平台上共有16种IRQ)。request_irq接收IRQ编号、函数名称、标志、/proc/interrupts的名称以及传送给中断处理函数的一个参数。这里提到的标志包括 SA_SHIRQ和SA_INTERRUPT，前者表明用户愿意与其它中断处理程序分享 IRQ(通常是因为同一个 IRQ上有多个硬件设备)；而后者表明这是个高速中断。只有当这个 IRQ上还没有处理程序，或者两者都愿意共享这个IRQ时，request_irq函数才会成功。

接下来，我们从中断处理程序内部与硬件进行通信，并使用 tq_immediate和mark_bh(BH_IMMEDIATE)调用queue_task_irq，以调度底半处理。在版本2中我们不能使用标准的queue_task，这是因为中断有可能正好发生在其它的 queue_task中间。我们之所以需要使用mark_bh，是因为以前版本的Linux只有一个由32个底半处理所组成的队列，而现在它们中的一个(BH_IMMEDIATE)已经被用作底半处理的链接列表，这是为那些没有得到分配给它们的底半处理入口的驱动程序所准备的。

Intel体系结构的键盘

警告 本章余下的内容完全是与Intel相关的。如果读者不是在Intel平台上运行，这些内容将不能正常工作。读者甚至不要去编译这些代码。

在写本章的示例代码时我遇到了一个问题。一方面，为了让所给出的例子能有用，它必须可以在所有人的计算机上运行，且能得到有意义的结果。但是另一方面，内核中早已经为所有的通用设备准备了设备驱动程序，而那些设备驱动程序与我将要编写的驱动程序是不能共存的。最后我找到了解决的办法，就是为键盘中断写驱动程序，并且首先关闭常规的键盘中断驱动程序。因为它被定义成内核源文件 (特别是指drivers/char/keyboard.c)中的静态符号，所以没有办法恢复它。如果用户珍惜自己的文件系统的话，在执行 insmod命令插入这个代码以前，请先在另一个终端上执行 sleep 120;reboot。

该代码把它自己绑定在IRQ 1上，在Intel体系结构下，IRQ1是控制键盘的IRQ。这样，当它接收到键盘中断时，它读键盘的状态 (在程序中使用 inb (0x64)来实现)，并查看代码，该代码是由键盘所返回的值。然后在内核认为适合的时候，它立刻运行 got_char，该函数将给出所使用的键的代码 (即查看到的代码的前七位)，并给出该键是被按下 (如果第八位为 0)还是被松开(如果第八位为1)。

```
ex intrpt.c

/* intrpt.c - An interrupt handler. */


/* Copyright (C) 1998 by Ori Pomerantz */



/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/tqueue.h>

/* We want an interrupt */
#include <linux/interrupt.h>

#include <asm/io.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
```

```
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Bottom Half - this will get called by the kernel
 * as soon as it's safe to do everything normally
 * allowed by kernel modules. */
static void got_char(void *scancode)
{
  printk("Scan Code %x %s.\n",
    (int) *((char *) scancode) & 0x7F,
    *((char *) scancode) & 0x80 ? "Released" : "Pressed");
}
/* This function services keyboard interrupts. It reads
 * the relevant information from the keyboard and then
 * schedules the bottom half to run when the kernel
 * considers it safe. */
void irq_handler(int irq,
                 void *dev_id,
                 struct pt_regs *regs)
{
  /* This variables are static because they need to be
   * accessible (through pointers) to the bottom
   * half routine. */
  static unsigned char scancode;
  static struct tq_struct task =
        {NULL, 0, got_char, &scancode};
  unsigned char status;

  /* Read keyboard status */
  status = inb(0x64);
  scancode = inb(0x60);

  /* Scheduale bottom half to run */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  queue_task(&task, &tq_immediate);
#else
  queue_task_irq(&task, &tq_immediate);
#endif
  mark_bh(IMMEDIATE_BH);
}



/* Initialize the module - register the IRQ handler */
int init_module()
{
```

```
    /* Since the keyboard handler won't co-exist with
     * another handler, such as us, we have to disable
     * it (free its IRQ) before we do anything. Since we
     * don't know where it is, there's no way to
     * reinstate it later - so the computer will have to
     * be rebooted when we're done.
     */
    free_irq(1, NULL);

    /* Request IRQ 1, the keyboard IRQ, to go to our
     * irq_handler. */
    return request_irq(
    1,  /* The number of the keyboard IRQ on PCs */
    irq_handler,  /* our handler */
    SA_SHIRQ,
    /* SA_SHIRQ means we're willing to have other
     * handlers on this IRQ.
     *
     * SA_INTERRUPT can be used to make the
     * handler into a fast interrupt.
     */
    "test_keyboard_irq_handler", NULL);
}


/* Cleanup */
void cleanup_module()
{
  /* This is only here for completeness. It's totally
   * irrelevant, since we don't have a way to restore
   * the normal keyboard interrupt so the computer
   * is completely useless and has to be rebooted. */
  free_irq(1, NULL);
}
```

第12章　对称多处理

要提高硬件的性能，最简单的(同时也是最便宜的)方法是把多个CPU放到一个主板上。实现这一点，可以有两种方法。一是使不同的CPU执行不同的任务(即非对称多处理)，或者使这些CPU并行运行，执行同样的任务(即对称多处理，简写为SMP)。进行非对称多处理非常需要对计算机将要执行的任务有特别的了解，而在像Linux这样的操作系统中，这是不可能的。另一方面，对称多处理相对要容易实现一些。这里所说的“相对容易”就是相对的意思——并不是指它真的容易实现。在对称多处理环境中，CPU共享同一个内存，这样做的后果是一个CPU中运行的代码可能会影响另一个CPU所使用的内存。用户不再可以肯定自己在前面一行中设置了值的那个变量仍然保持着那个值——另一个CPU可能在用户不注意的时候对那个变量进行了处理。很显然，要编出这样的程序是不可能的。

在进程编程时，一般来说上面这个问题就不是什么问题了，因为进程在某个时刻一般是只在一个CPU上运行的。而另一方面，内核则可以被运行在不同CPU上的不同进程所调用。

在版本2.0.x中，这个也不是什么问题，因为整个内核就是一个大的自旋锁(spinlock)。这意味着如果某个CPU在内核中而另一个CPU试图进入内核(假设是由于系统调用)，则后到的那个CPU必须等待，直到第一个CPU处理完，这使得Linux SMP比较安全，但是效率却相当低。

在版本2.2.x中，几个CPU可以同时位于内核中，这是模块编程人员需要留意的地方。我已经就SMP的问题向其它高手求助了，希望在本书的下一个版本中将会包含更多的信息。

第13章　常 见 错 误

在读者踌躇满志地准备动手编写内核模块以前，我还要提醒大家注意一些事情。如果是因为我没有警告过你而发生了不愉快的情况，请把你遇到的问题告诉我。我将把你买此书所付的钱全部还给你。

1) 使用标准库　不能这样做，在内核模块中用户只能使用内核函数，也就是可以在/proc/ksyms中找到的那些函数。

2) 关闭中断　用户可能需要在短时间内暂时关闭中断，那没什么问题，但是事后如果忘了打开它们，系统将会瘫痪，用户将不得不把电源拔掉。

3) 无视危险的存在　可能不需要提醒读者，但是最后我还是要说，只是为了以防万一。

附录A　2.0和2.2之间的差异

实际上我对整个内核了解得并不是很透彻，没有透彻到能够列出所有变化的地步。在转换本书例子的过程中(或者更确切地说是对Emmanuel Papirakis所做的转换进行修改)，我遇到了下面的差异，我把它们全部列举了出来，以便帮助模块编程人员 (特别是那些曾经学习过本书的前一版本，并熟悉我所使用的技术的读者)转换到新的版本。

希望进行转换工作的用户还可以访问如下网址：

http://www.atnf.csiro.au/~rgooch/linux/docs/porting_to_2.2.html.

1) asm/uaccess.h　如果需要使用put_user或者get_user，用户必须用#include包含这个文件。

2) get_user　在版本2.2中，get_user既可以接收指向用户内存的指针，也可以接收内核内存中的变量，以便填充用户信息。之所以这样，是因为在版本 2.2中get_user可以一次读两个或四个字节，如果我们所读的变量是两个字节或四个字节长的话， get_user可以读入它。

3) file_operations　该结构已经在open函数和close函数之间加入了一个刷新函数。

4) file_operations中的close函数　在版本2.2中，close函数返回一个整数，所以它可以失败。

5) file_operations中的read和write函数　这两个函数的头部已经改变了。在版本 2.2中，它们不再返回整数，而是返回一个 ssize_t类型的值，它们的参数列表也不同了。索引节点不再作为一个参数，但同时却加入了文件中的偏移量作为参数。

6) proc_register_dynamic　该函数已经不再存在了。取而代之的是，用户可以调用proc_register并把结构的索引结点字段置为0。

7) 信号　在任务结构中的信号不再是32位长的整数值，而是变成了由 _NSIG_WORDS整数组成的一个数组。

8) queue_task_irq　即使用户希望从中断处理程序内部调度任务来执行，他也应该使用queue_task，而不要使用queue_task_irq。

9) 模块参数　用户不再是只把模块参数定义为全局变量了。在 2.2中，用户必须同时使用MODULE_PARM来定义它们的类型。这是一个很大的改进，因为它允许模块可以接收以数字开头的字符串参数，而不会搞混淆。

10) 对称多处理　内核不再是一个大的自旋锁了，这就意味着内核模块在使用 SMP时必须小心。

附录B 其他资源

如果笔者愿意的话，可以轻易地在本书中再加入几章。可以加入一章介绍如何创建新的文件系统，或者介绍如何增加新的协议栈 (但这是不必要的 —— 因为读者很难找到Linux所不支持的协议栈)。当然还可以加入一些内核机制的解释，而这些机制我们并没有接触过，例如引导机制或者磁盘接口。

然而，笔者没有这么做，因为笔者写这本书的目的是为了探索内核模块编程的奥秘，希望教给用户一些内核模块编程的通用技术。对于那些对内核编程有着强烈兴趣的读者，笔者建议他们阅读如下网址的内核资源列表： http://jungla.dit.upm.es/~jmseyas /linux/kernel /hackers-docs.html。正如Linus所说的那样，学习内核的最佳方法就是自己阅读代码。

如果读者希望得到更多的短内核模块的例子，我建议阅读 Phrack杂志，即使用户对安全性不是太感兴趣，作为一个程序员也应该培养这方面的兴趣。 Phrack杂志上的内核模块都是一些很好的例子，介绍了用户可以在内核中所做的有关工作。而且这些例子都很短，读者不用费太大的劲就可以弄懂它们。

希望我能帮助读者成为一个更好的程序员，或者至少培养了在这方面的兴趣。如果读者写出了有用的内核模块，希望也能按照 GPL把它们出版出来，这样我也可以使用它们。

附录C　给出你的评价

这是一本“自由”的书。在GNU公共许可证所规定的内容以外，读者不受任何限制。如果读者想为这本书做点什么，我有以下几点建议：

- 给我寄一张明信片，地址是：

  Ori Pomerantz

  Apt. #1032

  2355 N Hwy 360

  Grand Prairie, TX 75050

  USA

  如果希望收到我的致谢回函，请在明信片上写清你的电子邮箱地址。
- 给自由软件组织捐献一些钱，或者一些时间 (更佳)。编写一个程序或者写作一本书，并按照GPL的条款出版，教别人怎样使用自由软件，例如 Linux或者Perl。
- 向别人解释一下自私与社会生活是格格不入的，与帮助其它人是格格不入的。我很高兴我写了这本书，并且我相信出版这本书将来一定会获得回报。同时，我写这本书是为了帮助读者(如果我做到了的话)。记住，快乐的人常常比不快乐的人对自己更有用，而有才华的人常常比低能儿要有帮助的多。
- 高兴起来。如果我将遇见你，而且你是个愉快的人，这次会面将会给我留下美好的印象，而且愉快的个性也会使你变得对我更有用。